後乗せページによる効率的なログ構造化インデックス

上 野 康 平[†1]　笹 田 耕 一[†1]

本論文では，後乗せページという新しい構造を導入することによって，ログ構造化データ構造における木構造インデックスの取り扱いを改善できることを示す．ログ構造化データ構造における更新は，新しいデータを含むページ (ログ構造の構成単位) をログ状の構造の末尾に追記することによって行う．今までのログ構造化データ構造では，古いページを参照していたページが存在していた場合，それらも新しいページを参照するように再帰的にログ構造に書き出される必要があった．この更新プロセスでは，階層的な参照関係を持つ木構造のインデックスを使用した際に，木のルートまで参照更新の伝搬処理が発生してしまい，大量のページが参照の更新のために再度書き出されることになってしまう．今回，この問題に対し，後乗せページという特殊なページを導入することによって，更新伝搬を抑制することに成功した．後乗せページには，通常のページとは違い，更新元への参照が含まれている．更新元ページと後乗せページの関係を変換表を用いて管理することにより，古いページに対する参照を，適宜更新データを持つ後乗せページへの参照と読み替えることができる．以上の後乗せページを用いた更新を用いることで，木構造インデックスを取り扱った際に発生する参照更新の伝搬を遅延することができる．本論文では，提案手法であるログ構造化データ構造への後乗せページの導入，及びこれを扱う実装について詳しく述べる．さらに，提案手法を導入することで，木構造インデックスを扱う際に性能向上が得られることを評価により示す.

Override Pages for Efficient Log-Structured Index

Kouhei Ueno[†1] and Koichi Sasada [†1]

In this paper, we introduce *override pages*, a simple expansion to log-structured data structures which greatly improves efficiency of tree indices. In log-structured data structures, updates are performed by appending a page, an element that composes log structure containing update data, to tail of the log. In previous methods, all references to the page that contained old data had to be also recursively updated and rewritten to tail of the log. However, in this process, an single update of indices with hierarchical references would result in massive rewrite of pages, caused by reference updates propagating to the tree root. We have relieved this issue by introducing *override pages*, special pages specifying update while retaining old references. Override pages, unlike ordinary pages, contain a reference to the page containing old data. By keeping track of these relations between the old page and a new override page, it is possible to redirect references to old pages to the override pages containing new version of the data. By updating indices with override pages, we could delay the reference update propagation, and therefore greatly reduce the number of written pages in a single update. In this paper, we first describe our model of override pages, and design and implementation when incorporating it to log-structured data structures. Furthermore, we evaluate performance improvement in tree indices achieved by override pages.

1. はじめに

多くのデータベースシステム (DBMS) において，レコードの不揮発性ストレージへの読み書き処理は複数のストレージレイヤにより抽象化されている[1)](図 1). ユーザに対して，レコードはテーブル，XML，ドキュメントといった形で提供されるが，これらを扱う論理的データ構造は物理的データ構造の上のレイヤとして提供される．物理的データ構造は，Key, Value 組として論理的データ構造から提供される多数の索引キー付き可変長データを，一次元バイト列としてどのように不揮発性ストレージ上に保存するかを規定する．

物理的データ構造は不揮発性ストレージ上に格納されるため，メモリ上で使われるデータ構造とは異なった性質が求められる．まず，不揮発性ストレージは 2^8 から 2^{16} バイト単位の読み書きに最適化されているため，メモリとは異なり読み書き粒度は大きい．さらに，アクセス速度もメモリに比べて遅いため，多少 CPU の計算コストが増えても，アクセスの回数を削減する

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

ⓒ 2011 Information Processing Society of Japan

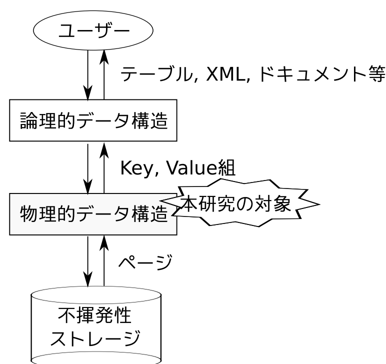

図 1 DBMS におけるストレージレイヤの抽象化

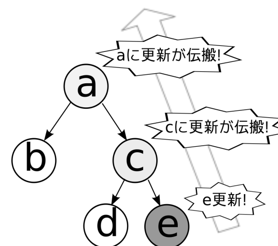

図 2 木構造インデックスでの更新の伝搬．それぞれの円はインデックスの節を格納しているページを示す．ログ構造化データ構造では，ページ e の更新がこれを参照しているページ c，a にも伝搬する．

ことが重要になる．

我々は，物理的データ構造として，Rosenblum らによって考案されたログ構造化データ構造[2)3)] に注目した．ログ構造化データ構造とは，データの変更ログをそのままデータ構造として用いる手法である．ログ構造化データ構造は，ページと呼ばれる構造の集まりからなり，レコードやインデックスはこのページの中に保存される．ページに含まれるデータの更新は，元のページの上書きではなく，新しいデータを含むページを変更ログへ追記することによって行う．

追記による更新は，ログ構造化データ構造の書き込み性能や堅牢性に貢献している．まず，ログへの追記は逐次アクセスにより行われるため，シークが発生せず，ディスクのバンド幅を有効に活用することができる．また，更新データは以前のデータを破壊的に上書きするのではなく，ストレージ上の別の場所に書き込まれるため，更新処理の発生中でも終了を待たずに読み込みを行える．さらに，更新処理中の事故で処理が停止しても，更新前のデータをまとめて失ってしまうことはない．

ログ構造化データ構造には，実際のレコード列に加え，最新のデータを参照する木構造のインデックスが共に格納される．例えば，ログ構造化データ構造を用いている NILFS[4)] や CouchDB[5)] といったシステムでは，BTree のインデックスを用いている．

しかし，ログ構造化データ構造を実用化する上で，この木構造型のインデックスの更新に大量の書き込みが必要となるという欠点があった．ログ構造化データ構造では，データは更新ごとに新しいページへ格納される．そのため，データ本体を格納しているページの更新に伴い，それを参照していたページにも更新が発生する．しかし，この参照の更新も，ページの移動を伴うため，ページを間接的に参照していたページにも更新が伝搬していく．木構造インデックスは，レコード列と同様にログ構造化データ構造中のページに格納される．レコードの更新が発生すると，最新のレコードを持つページを指し示すために，インデックスの更新が付随して発生する．木構造インデックスの構成ページは階層的な参照関係を持つため，この更新はルートまで伝搬していき，ルートまでの大量のページ更新につながる (図 2)．

これは，ファイルシステムなどレコード長が大きく，かつまとまった更新が行われるシステムではあまり問題にならないが，RDBMS のような細かなデータ更新が発生するようなシステムにおいて，深刻な性能低下をひきおこす．このようなシステムでは，更新データは 1 ページに収まるほど小さなものが多い．しかし，この 1 ページを更新するために，ページの場所変更に伴うインデックス更新の伝搬が発生してしまい，結局大量のページを書き出す必要がでてきてしまう．これは，実際に更新が発生した少数のページのみを更新する上書きを前提としたデータ構造に比べて，一回の更新に必要な書き込み量が増えてしまう．

我々は，**後乗せページ**という特殊なページをログ構造化データ構造に導入することにより，この問題を解決した．後乗せページは，更新データの他に，更新の対象となる元ページへの参照を持つ．これをもとに，以後の元ページへの参照は，後乗せページへの参照へと読み替えられる．後乗せページを用いることで，古い参照を持つインデックスの更新伝搬を抑える事ができ，毎回木のルートまでインデックスを書き出す必要がなくなり，書き込み性能の向上を実現することができる．

本論文では，ログ構造化データ構造に対して提案手法である後乗せページを導入することで，木構造インデックスを扱う際に問題となるページ更新の伝搬を抑制できることを示す．まず，前提になるログ構造化データ構造について，また本手法を導入しなかった場合，木構造インデックスを扱う上でどのような問題が発生しているのか紹介する (2 章)．次に，提案手法である後乗せページによりこの問題がどのように解決するのか述べる (3 章)．さらに，この後乗せページを用いたログ構造化データ構造の管理手法について詳細を述べる (4 章)．また，この拡張により，ログ構造化データ構造におけるインデックスの書き込み性能が向上す

ⓒ 2011 Information Processing Society of Japan

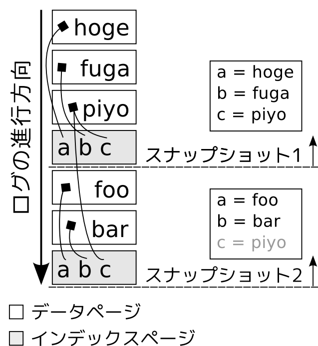

図 3 ログ構造化データ構造における更新処理

ることを，定量的な評価によって確かめる (5 章)．最後に，この研究の関連研究について述べ (6 章)，まとめる (7 章)．

2. ログ構造化データ構造

提案手法は，ログ構造化データ構造に対する拡張である．ここでは，ログ構造化データ構造についての概要と，木構造インデックスと共に用いた際に，ナイーブな実装ではどのような問題が起きるのかを解説する．

2.1 概　　要

ログ構造化データ構造とは，データのログをそのままデータの格納場所として用いる物理的データ構造である．ここでは，提案手法の対象とするログ構造化データ構造が，一般的にどのように実装されているのかを述べる．

このログは，ページと呼ばれる 2^8 から 2^{16} バイトの構造の集まりからなる．これは，物理的な不揮発性ストレージの読み書き単位であるセクターやブロックに一対一対応している．ページはその内容により，データ本体が格納されているページとインデックスが格納されているページの二種類に分類される．データページには，Key, Value 組が格納されており，インデックスページにはこれらのデータが格納されているページへの参照が格納されている．手法によっては，これらの他に管理情報を格納するページを用いる場合もあるが，今回のモデルではそれらは扱わない．

これらのページはログ上に線形に保存され，上書きは許されない．ページはその種類に関わらず，原則として更新された順番に保存される．ログ構造化データ構造における古いデータの更新は，ログへ新しいデータを持つページを追記することによって行われる．これは，古いデータが格納された領域に新しいデータを上書きすることにより更新する，一般的な上書き型のデータ構造とは対照的である．

ログ構造化データ構造におけるデータの更新は，更新先データの書き込み，データへの参照の書き直し，の二段階で行われる (図 3)．まず，更新先のデータを含むページが最後のログの直後に追記される．例えば，Key `a`，`b` に関連づけられる Value をそれぞれ `foo`，`bar` に更新する場合，まず更新先の Value `foo`，`bar` を含むページがログに追記される．次に，元データへの参照を含む全てのページが，再度書き出される．この際，更新元のデータの参照は，全て更新先のデータを指すように書き換えられる．図 3 の場合，インデックスページが古い `a`，`b` の内容である `hoge`，`fuga` を参照しているため，新しい参照先 `foo`，`bar` を指すように更新され，再度書き出される．さらに，これらの更新されたページに依存していたページが存在していた場合，それらのページも再帰的に書き出される．

ログ構造化データ構造では，スナップショットを簡単に扱うことができる．スナップショットは，任意の更新処理を行った直後の状態のレコード列にアクセスするための機能である．ログ構造化データ構造において，スナップショットはある更新処理において最後に書き出されたページの位置を保存することで，簡単に実現することができる．あるスナップショットからレコードを取り出す際には，その保存された位置よりも前に保存されたインデックスを用いてレコードを探索するだけで良い．

一見，この追記による更新は上書きによる更新に比べ無駄が多いように見える．しかし，以下に述べるような様々な利点がある．

まず，書き込み済みのデータは常に一貫した状態で存在しているため，書き込み処理の途中でも，古いデータに対しては読み込みを行うことができる．上書き型データ構造で必要だった排他制御は不要となる．さらに，任意の地点のスナップショットへの参照も，以前のインデックスを基にデータを探索することで，簡単に実現できる．

さらに，更新処理中にシステムが停止しても，そこまでのデータが破損することはなく，最後に書き込まれた以前のデータは確実に読み出すことができる．ARIES アルゴリズム[6] のような別途の先行書き出しログを用いた復旧プロセスは一切必要なく，最後に書き出されたインデックスを辿るだけで有効なデータを取得できる．

最後に，任意の区間のデータの更新ログを高速に取得することができる．データの格納されているページはその更新順に記録されているため，任意の二つの時点間の差分データは，この線形の構造の一部として簡単に取り出すことができる．

2.2 木構造インデックスを用いた場合の問題

以上で述べたように，ログ構造化データ構造は物理的データ構造として好ましい性質を備えている．しかし，木構造のインデックスと共に用いると大幅な性能低下が生じるという問題がある．

ログ構造化データ構造において大量のデータを扱う

ⓒ 2011 Information Processing Society of Japan

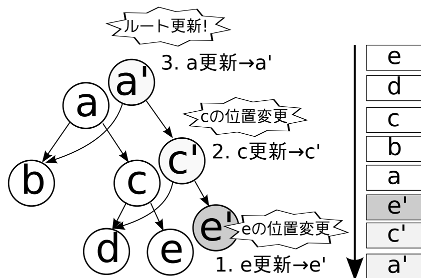

図 4 ログ構造化データ構造における木構造インデックスの更新．それぞれの円はインデックスの節を格納しているページを示す．ページ e の内容更新により，新しい内容は別ページ e’ に格納されるため，この新しいページに参照を置き換える為に c → c’, a → a’ の更新が付随して発生する．

際，効率的な探索のためにインデックスには BTree 等のデータ構造が用いられる．これらの木構造のインデックスは，全て親から子へとつながる階層的な参照関係を持つ．

木構造のような階層的な参照関係をログ構造化データ構造で扱う際，一回の更新に伴って書き出されるページの量は増えてしまう (図 4)．ページの更新の際に親の節ページまで参照関係の更新が伝搬するため，途中にある節ページ全てを新しいページとしてログに書き出す必要がある．特に，節ページにある程度余裕が確保されている BTree などの構造では，子の変更によって親の節ページの内容に変更が必要なことはスプリットなど特別な場合を除いては無く，上流の節ページはこの参照関係の更新だけのために毎回書き出されることになってしまう．これは，子のページの更新を上書きによって済ませれば良い上書き型の手法に比べ明らかなオーバーヘッドになる．

3. 提　　案

今回，我々は後乗せページという参照関係を崩さず更新を行える特殊なページを導入することで，この問題を解決した．

後乗せページは，我々が新たに導入した，データの更新を示すページである．後乗せページには，通常のページと同様，Key, Value 組やインデックスの更新先データを格納するが，これに加え，どのページに対する更新なのかを示す，更新元のページへの参照を格納する．この更新元ページの参照と後乗せページの対応をメモリ上に構築する変換表を用いて適切に管理することにより，更新元ページへの参照が発生した際に，新しいデータを持つ後乗せページを代わりに参照するように，読み替えることができる．なお，不揮発性ストレージからの読み込み時に，変換表は存在しない為，後乗せページに含まれる更新元参照の情報を基に変換表を復元する．

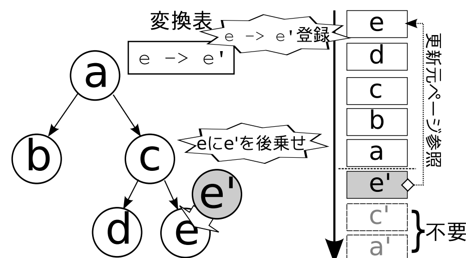

図 5 後乗せページを用いた木構造インデックスの更新．それぞれの円はインデックスの節を格納しているページを示す．ページ e を後乗せページ e’ を用いて更新することで，c の持つ古い参照でも e’ の更新内容の読み取りが可能になるため，ページ c, a に更新が伝搬しない．後乗せページを用いない場合 (図 4) と比較して書き込み量が抑えられていることがわかる．

このように，古いページ参照でも更新データを読み込めるようにすることで，ページ参照の更新を遅延することができる．しかし，後乗せページを数や有効範囲の制限なしに用いると，いくつか問題が発生する．

一つは，変換表の大きさの問題である．変換表にあまり大量のエントリを格納することはリソース利用の観点から望ましくない．さらに，この変換表はページ読み込み指示の度に参照されるため，実行速度の観点からも少数のエントリに留めておく必要がある．

もう一つは，最新データの読み込みにかかるコストの問題である．通常のログ構造化ストレージであれば，最新データを参照する際は，最新のインデックスにより参照されているページを読み込むだけで済む．しかし，後乗せページが導入されている環境では，変換表を見て，参照先ページに対する後乗せページの有無を調べなければならない．もし後乗せページの有効範囲に制限がない場合，ログの最初の方にある後乗せページの存在も考慮しなくてはならず，結果としてログ構造全体を読み込む必要が出てくる．

以上の問題を防ぐ為に，後乗せページによる古いページ参照の転送設定を一定期間で無効化する．後乗せページが一定量に達すると，改めて依存関係の整理が行われ，上流の節ページは直接この後乗せページを指すように参照が書き換えられる．この処理のことを以後，リベース処理と呼ぶ．リベース処理により，リベース処理以前の変換表による参照の読み替えなしに最新のページ内容が参照できることが保証される．これにより，変換表のサイズの増大を防ぐのと同時に，最新データが必要になった際にログ全体を読み込まなくても済むようにしている．リベース処理の詳細は，4 章にて詳しく述べる．

この後乗せページを導入することで，木構造インデックスにおける更新の伝搬処理を遅延，抑制することができる (図 5)．深い階層的な参照関係を持つ木構造インデックスのページを更新する際，後乗せページを用いると，上流の節ページの参照情報更新をする必

ⓒ 2011 Information Processing Society of Japan

表 1 ログ構造化データ構造の操作一覧

put(k, v) → s	Value v を，Key k に新しく結びつける． 操作適用後のスナップショットを示す ID s を返す．
get(k[, s]) → v	指定された Key k に対応する Value v を返す． スナップショット ID s が指定された場合は， Value をそのスナップショットより取得する．

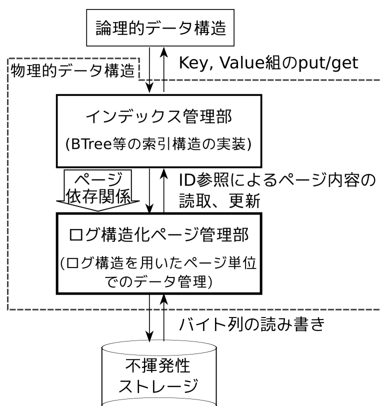

図 6 ログ構造化データ構造による物理的データ構造実装の構成

要がなくなる．このように古い参照情報を持ったままでも，変換表により古いページ参照が適切に転送されるため，更新先のページを正しく参照することができる．上流の節ページの参照情報の更新は，リベース処理が起こるまで遅延される．この際，複数の参照情報の更新をまとめて行うことが可能なため，書き出されるページ数は後乗せページなしで毎回更新を伝搬した場合に比べて削減される．

4. 実　　装

ここでは，後乗せページを導入したログ構造化データ構造を実装する手法について述べる．

4.1 インターフェース

本実装では，物理的データ構造の実装としてログ構造化データ構造を実装した．ユーザは，本実装を用いることによって，レコードの読み書き操作を行う事ができる．具体的には，ユーザに対して表 1 の操作をサポートする．これら操作の実装については，節 4.4, 4.5 にて詳しく述べる．

4.2 全 体 構 成

本実装は，インデックス管理部とログ構造化ページ管理部の 2 つの部分からなる (図 6).

インデックス管理部は，Key, Value 組を効率的に探索するための木構造インデックスの実装である．レコードの読み書きが効率的に行えるように，Key に基づくインデックス構造や，Key, Value 組が格納されるデータページ中の構造を管理する．インデックス管理部の実装からは，ページがログ構造で管理されていることは隠蔽されており，ログ構造内のページ管理は，全てログ構造化ページ管理部を介して行う．

このようにインデックス管理部が分離できるのは，本提案手法ではインデックスアルゴリズムに対する制約が緩いためである．本実装では，インデックスのアルゴリズムとして BTree を用いているが，一般的にログ構造化データ構造で用いる事が可能なインデックス構造であれば，何でも使うことができる．例えば，BTree の亜種や，ハッシュテーブルなどの構造を用いる事もできる．

ログ構造化ページ管理部は，ログ構造中のページ管理を行う．不揮発性ストレージ上に構築されるログ構造を管理し，インデックス管理部から指示されたページ操作を行う．ここで行われる処理の詳細について，次節より詳しく述べる．

4.3 ページの構造

本実装では，1 ページは 4096 バイトの構造体として定義している．それぞれのページは，以下のようなヘッダーを持つ．

```
struct page_hdr_t {
  page_id_t id;
  page_type_t type;
  page_id_t idOvrTgt;
};
```

このうち，`id` はページに割り当てられる ID 番号，`type` はページの内容の種類を示すフィールドである．`idOvrTgt` は本提案手法である後乗せページに特有なフィールドで，後乗せページが更新対象とするページへの参照を，その ID 番号によって行う．新規データによるページの場合は，更新元のページが存在しないため，`idOvrTgt` は値`-1` を持つ．

4.4 get の実装

本実装では，get 処理 (Key に対応する Value の取得処理) は以下の手順で行う．

まず，インデックス管理部は，Key, Value 組の格納されているデータページを特定する．Key, Value 組の探索は，木構造を探索することによって行う．例えば，BTree では，ルートから探索を開始し，key の該当範囲を担当する子のページへの参照を辿っていく．この際生じるページの読み込み処理は，ログ構造化ページ管理部にページ ID を指示することによって行う．

ログ構造化ページ管理部は，ページ ID によりインデックス管理部から指定されたページの内容を読み出す．ログ構造化データ構造では，ページの ID 番号とログ構造中の格納順は一致している．そのため，後乗せページを考えない場合，ページの場所は単純なポインタ演算で求めることができる．

提案手法では後乗せページによるページ参照の書き換えを許しているため，参照先のページ ID に対する読み替えが存在するか調べる必要がある．この調査は，変換表を調べることによって行う．変換表は，後乗せ

ⓒ 2011 Information Processing Society of Japan

ページによるページ ID の読み替え指示の一覧を保持する．変換表の実装は，変換元ページ ID，後乗せページ ID，後乗せページ ID のスナップショット番号の 3 項目のエントリをもつリングバッファである．変換表がリングバッファとして実装されている都合上，本実装では遡れるスナップショットの範囲が限定されている．これは，本実装の制限である．

変換表の有効なエントリの内に，参照先ページ ID に対する後乗せページの読み替えの指示が見つかった場合，その後乗せページを代わりに読み込む．この際，有効な読み替え指示は，スナップショット番号が get 時に指定された以前のものに限定される．これにより，後のスナップショットで変更された後乗せページが以前のスナップショット参照時により読み込まれることを防ぐ．

4.5 put の実装

本実装では，put 処理 (Key,Value 組の挿入処理) は以下の手順で行う．

まず，インデックス管理部は，Key, Value 組の格納されるべきデータページを特定し，データページの更新データを作成する．この際発生するインデックスページやデータページの取得処理は，節 4.4 に示した get の実装と同様である．インデックス管理部は，そのデータページに新しく Key, Value 組が格納されることによるページの更新データを作製し，それをログ構造化ページ管理部に送信する．この際，このデータページの参照関係もログ構造化ページ管理部に共に通知する．

ページの更新指令を受けたログ構造化ページ管理部は，新しいページを作製し，インデックス管理部に指示された更新内容を保存する．通常の実装では，上流のインデックスページに対して，新しいページへの参照の書き換えが必要になるが，本実装では後乗せページを用いて更新を行い，上流のインデックスページの参照更新を遅延する．

後乗せページの更新元参照 `idOvrTgt` の参照先は，指示された更新元ページを指すが，更新元ページが有効な後乗せページであった場合はその更新元参照を引き継ぐ．この更新元参照の引き継ぎは，変換表のエントリ数を削減するために行う．もし引き継ぎがなかった場合，元後乗せページに対する新後乗せページの更新関係をさらに管理する必要が発生する．

最後に，リベース処理用の情報の保存を行う．ログ構造化ページ管理部では，古い内容を持つページへ参照しているページ一覧を，その参照関係と共に管理している．今回，更新データと共にインデックス管理部より通知された古いページの情報を，この一覧に追記する．ここで保存される情報の利用については，次節にて詳しく述べる．

4.6 リベース処理

本実装では，3 章で述べたように，変換表のエントリ数を抑え，最新データの読み込みコストを低減するために，後乗せページによる古いページ参照の読み替えを一定期間に制限している．リベース処理は，変換表エントリによる読み替え指示を失効し，参照の更新として上流ページに反映する．

リベース処理は，この参照が古くなったページ数が一定量に達するか，後乗せページによる読み替えの数が一定量以上になった場合に発生する．これにより，一度に有効な後乗せページの量を制限している．

リベース処理では，後乗せページにより遅延されていた参照更新の伝搬処理を行う．提案手法では，このようにリベース処理により参照更新の伝搬を一回にまとめて行うことで，更新に伴う書き出しページ量の大幅な削減を実現する．

インデックス管理部は，ページ更新により参照が古くなったページをログ構造化ページ管理部に通知する (4.5 節参照)．具体的には，古くなったページ ID 及びその古くなったページ自身を参照しているページ ID のペアを通知する．

ログ構造化ページ管理部は，この通知された情報を基に，末端に近い節を含むページから書き出していく．先に下流の節を書き込むことで，下流の節の位置変更に伴い再度上流の節を更新する必要を避けている．

リベース処理は，参照の更新，変換表の失効の二段階で行う．

まず，古いページ参照を含むページに対し，最新のページを指すように参照の更新を行う．依存情報に基づき，古いページ参照を含むページ一覧に対してトポロジカルソートを行い，末端に近い順番のページ ID のリストを得る．次に，リストに含まれるそれぞれのページに含まれる古いページ参照を，変換表を用い更新する．この更新も，新しいページの追記によって行われるため，そのページを参照しているページも再帰的に更新していく．

次に，変換表の内容を失効する．リベース処理により，それ以前の後乗せページによる読み替えなしに最新データへのページ参照が機能する為である．これは，リベース処理のスナップショット番号を保存し，get 時にそれ以前のスナップショット番号を持つ変換表エントリを参照しないことによって行われる．

5. 評　　価

提案手法の有効性を評価するため，提案手法によるログ構造化データ構造の実装を用いて評価を行った．実装は，C++を用いて行った．ソースコードの行数はコメントを含め 2000 行程度である．また，評価環境は，表 2 に示す通りである．

5.1 後乗せページによる書き出しページ数の削減効果

提案手法である後乗せページを使用した場合と，使

ⓒ 2011 Information Processing Society of Japan

表 2 評価環境

OS	Linux 2.6.32-5-amd64
CPU	Intel Xeon E5345 @ 2.33GHz x 2
メモリ	DDR2-667 FB-DIMM 8GB
SSD	Intel SSDSA2MH160G2C1 (160GB MLC)

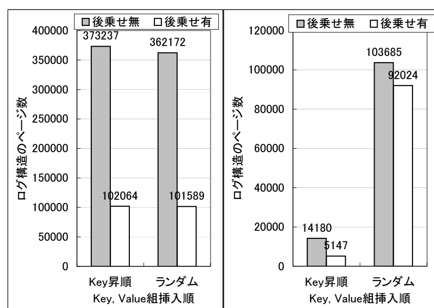

図 7 後乗せページによるページ使用量の削減効果 (右: 1 組ずつ put, 左: 30 組ずつ put)

用しなかった場合でページの使用量にどの程度差があるのかを調べた．評価は，10 万個の Key, Value 組を put した場合に，ログに保存されたページの数を計測することで行った．Key, Value 組の内容は，それぞれ 4 バイトの Key, 8 バイトの Value である．Key, Value 組の put 操作を行う順番として，Key の昇順に挿入した場合，及びランダムな順番で挿入した場合の二つのケースでテストを行った[⋆1]．また，put の粒度として，Key, Value 組を 1 組ずつ put した場合，30 組ずつまとめて put した場合それぞれの計測を行った．

計測結果を図 7 に示す．後乗せページの使用により，どのケースにおいてもページ使用量が削減できていることがわかる．

Key, Value 組を 1 組ずつ put したケースでは，挿入順にかかわらず 70%のページ量削減効果が得られていることがわかる．後乗せページを用いた場合，1 回の put 操作により変更されるページは節ページのスプリットの場合を除けば 1 ページになる．これに対し，後乗せページを用いない場合では，これらのページの更新が上流のページ（約 3 ページほど）に伝搬してしまい，これらを更新するためにより多くのページを消費している．

しかし，Key, Value 組を 30 組ずつ put したケースでは，Key 昇順に挿入した場合には 60%の削減効果が得られているのに対し，ランダムに挿入した場合では 10%程しかページ量削減効果が得られていない．後乗せページによるページ使用量削減は，上流ページに対する参照書き換えを遅延することにより行うが，ランダム 30 組ごとの put では，末端ページがスナップショット毎に 30 ページ変更されるため，書き込み処理における上流ページの書き換えが占める割合が少ない．よって，後乗せページによるページ使用量削減の効果が薄くなっている．

⋆1 それぞれ，DBMS におけるテーブルのプライマリインデックスとして用いた場合，セカンダリインデックスとして用いた場合を想定している．

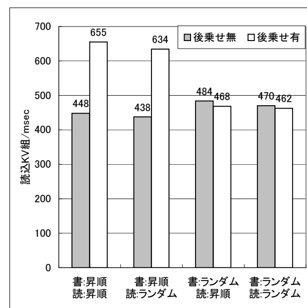

図 8 後乗せページを導入した場合の Key, Value 組 100 万件の読込スループットの変化

5.2 後乗せページによるオーバーヘッド

後乗せページ導入による読み込み処理のオーバーヘッドを計測した．Key, Value 組 100 万件のログ構造化データを用意し，全件を読み込む速度を測定した．Key, Value 組の内容は前節と同様である．挿入処理は 100 件ごとにまとめて put を行った．対象データとしては，Key の昇順に挿入したものと，ランダムに挿入したものを用意した．読み込み順としても，Key 昇順に読み込んだ場合，ランダムに読み込んだ場合の二つのケースでテストを行った．

計測結果を図 8 に示す．書き込みを Key 昇順に行ったデータでは，読み込む順番にかかわらず，後乗せページを用いた方がスループットが 45%ほど向上している．Key 昇順の書き込みでは，図 7(左) でわかるように，後乗せページの導入により総データ量が 6 割程度削減できている．この速度向上は，キャッシュ効率が向上していることによる．

次に，書き込みをランダムに行ったデータでは，後乗せページの導入によりスループットが 2〜3%低下している．書き込みをランダムに行ったデータでは，Key 昇順の場合と違い，後乗せページの導入による総データ量の削減効果は薄く，キャッシュ効率に変化はない．この低下は，変換表による後乗せページ管理のオーバーヘッドによる．

6. 関 連 研 究

後乗せや**差分パッチ**の管理によるデータベースの効率化の研究には，様々な既存研究がある．Severance らによる Differential File[7] では，大規模データベースに対して差分ファイルの管理手法に関して議論している．しかし，ここで述べられている差分ファイルは単純な変更レコードの集合であり，後乗せページのように構造化されておらず，変更元データの位置指定も

ⓒ 2011 Information Processing Society of Japan

ない．

Oh らによるログ構造化ファイルシステムの最適化の研究[8] は，更新の伝搬問題に言及している．Oh らは本手法に類似したインデックスの書き出しを遅延する Lazy Indirect Block Update を提案している．しかし，インデックスの書き出し前にシステムが停止してしまった場合の復旧は，その後のレコードの内容を全て走査することにより行っており，本手法に比べて確実性に欠ける．また，Oh らによる手法では，ログ構造化データ構造の特徴であるスナップショットによる参照ができない．

Wang らによる WOLF[9] は，ログ構造化データ構造の書き込み速度高速化手法である．ガベージコレクション (GC) による領域の再利用が行われるログ構造化データ構造に対して，新規データの書き込み位置を工夫することで，GC 処理による書き込みのオーバーヘッドを減らしている．これは，書き込み量を減らすことを目標とする我々の手法とは異なるアプローチである．

7. 今後の課題

今後の研究課題として，トランザクション処理への応用があげられる．2 章で述べたように，ログ構造化データ構造ではスナップショットを簡単に管理することができる．これを応用し，スナップショット分離環境下でのトランザクション処理基盤として本手法を拡張していく予定である．

また，後乗せページは，インデックスの構造に依存しない手法であるため，インデックスとして BTree 以外を用いることも可能である．特に，ハッシュテーブル等の木構造でないインデックスを用いた場合の本手法の有効性に関しては，まだ確認できていない．

最後に，今回の提案では，ログ領域の GC による再利用の実装や最適化がされていない．GC を実行した場合や，セグメント化による GC の最適化[3][9] を併用した場合の挙動に関しては，未評価である．

8. おわりに

本論文では，ログ構造化データ構造に対し後乗せページという特殊なページを導入することで，木構造インデックスを扱う際に生じていたページ更新の伝搬による問題を解決した．ログ構造化データ構造では，更新データを常に新しい位置に配置する．この為，今までのログ構造化データ構造では，更新対象となるページを間接的に参照しているページに関しても，参照の更新が伝搬していくという性質があった．これは，特に木構造インデックスを扱った場合に顕著であり，末尾のページの変更でも，ルートまで毎回書き出されなければならないという問題が発生していた．今回，後乗せページという新しい構造を導入し，参照関係を崩す事なくページ更新を行えるようにすることによって，この問題を解決した．後乗せページは，更新元のページへの参照を保持している．この情報を適切に管理することで，古いページへの参照を後乗せページへの参照と読み替えることができる．これにより，古いページ参照のままで最新データを参照できるようにすることで，参照の更新を抑制することができ，更新の伝搬を大幅に抑えることができた．本論文では，実際に後乗せページを導入したログ構造化データ構造の設計，実装を行った．評価により，後乗せページの導入は，わずかな読み込みオーバーヘッドで，書き込みデータ量を平均で 60%削減できることを示した．

参 考 文 献

1) Ramakrishnan, R. and Gehrke, J.: *Database Management Systems*, McGraw-Hill Education (2002).
2) Ousterhout, J. and Douglis, F.: Beating the I/O bottleneck: a case for log-structured file systems, *SIGOPS Oper. Syst. Rev.*, Vol.23, pp. 11–28 (1989).
3) Rosenblum, M. and Ousterhout, J.K.: The design and implementation of a log-structured file system, *ACM Trans. Comput. Syst.*, Vol. 10, pp.26–52 (1992).
4) 天海良治ほか: Linux 用ログ構造化ファイルシステム nilfs の設計と実装, 情報処理学会研究報告. [システムソフトウェアとオペレーティング・システム], Vol.2005, No.48, pp.61–68 (2005-05-25).
5) Katz, D.: CouchDB, `http://couchdb.apache.org/`.
6) Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H. and Schwarz, P.: Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, *ACM Transactions on Database Systems*, Vol.17, pp.94–162 (1992).
7) Severance, D.G. and Lohman, G.M.: Differential files: their application to the maintenance of large databases, *ACM Trans. Database Syst.*, Vol.1, pp.256–267 (1976).
8) Oh, Y., Kim, E., Choi, J., Lee, D. and Noh, S.H.: Optimizations of LFS with slack space recycling and lazy indirect block update, *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, New York, NY, USA, ACM, pp.2:1–2:9 (2010).
9) Wang, J. and Hu, Y.: WOLF–A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File System, *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, Berkeley, CA, USA, USENIX Association (2002).

ⓒ 2011 Information Processing Society of Japan